



2D platformer játék fejlesztése Unity keretrendszerben

Készítette

Bartus János

Programtervező informatikus Bsc.

Témavezető

Troll Ede

Tanársegéd

EGER, 2025

Tartalomjegyzék

Bevezetés	4
1. Technológiai áttekintés	6
1.1. Godot Engine	6
1.1.1. GDScript nyelv	6
1.1.2. A Godot verzióinak áttekintése	7
1.1.3. Licenszi kérdések	8
1.1.4. A motor fő erősségei	8
1.2. Unreal Engine	9
1.2.1. Blueprint	9
1.2.2. Unreal Engine verzióinak áttekintése	10
1.2.3. Licenszi kérdések	10
1.2.4. A motor fő erősségei	11
1.3. Unity	12
1.3.1. Programozási nyelv	12
1.3.2. Fontosabb verziók	13
1.3.3. Licenszelési kérdések	14
1.3.4. Főbb erősségei	14
1.3.5. Jelenetek	15
1.3.6. GameObject	15
1.3.7. Editor	17
2. Rendszerterv	21
2.1. A rendszer célja	21
2.2. Projektterv	21
2.3. Funkcionális terv	23
2.3.1. Rendszer szereplők	23
2.3.2. Rendszerhasználati esetek és lefutásaik	23
2.3.3. Menühierarchiák	24
2.4. Követelmények	25
2.4.1. Funkcionális követelmények	25
2.4.2. Nem funkcionális követelmények	26

2.5.	Használt fejlesztői eszközök	27
2.5.1.	Unity	27
2.5.2.	Github, Git	27
2.5.3.	Trello	27
3.	Saját projekt fejlesztése	28
3.1.	A platformer játék bemutatása	28
3.1.1.	A játékos karakter	28
3.1.2.	Ellenségek	29
3.1.3.	Pályarészek közötti átjárás	31
3.1.4.	Waypoint mutató	32
3.2.	Játékmechanikák és algoritmusok	32
3.2.1.	Mozgás és irányítás	32
3.2.2.	Támadások	33
3.2.3.	Pályarészek közti átjárás logikája	33
4.	Tesztelés	35
4.1.	Playtest	35
4.2.	Unit teszt	36
4.3.	Teszt jegyző könyv	36
4.4.	Összefoglaló	37
	Összegzés	38
4.5.	Bővítési terv	38
	Ábrák jegyzéke	39
	Irodalomjegyzék	40

Bevezetés

Gyerekkorom óta érdekeltek a videójátékok, már egészen kicsiként kezdtem el játszani. Az első komolyabb élményeimet apukám PlayStation 2 konzolján szereztem ahol sok időt töltöttem különböző játékokkal. Akkoriban még csak a játék öröme vonzott, nem is gondoltam volna hogy ennyire mélyen elmerülök ebben a világban. Ahogy idősebb lettem, általános iskola alsó tagozatában kezdtek jobban érdekelni a videójátékok nemcsak mint szórakozás hanem mint rendszerek. Elkezdtem azon gondolkodni hogyan működnek, mi és hogy irányítja a karaktereket és a játékon belüli eseményeket.

A programozás irányába a Minecraft vezetett. Felfedeztem benne a parancsblokkokat és azokkal próbáltam változtatni a játék világát. Egyre izgalmasabbá vált hogy a saját ötleteimet meg tudom valósítani, ekkor határoztam el hogy programozó akarok lenni.

A programozással középiskolában kezdtem el foglalkozni. Ekkor már tudatosan kerestem azokat a lehetőségeket, amelyekkel jobban megérthetem hogyan működnek különböző játékok és szoftverek. Az első lépéseimet kisebb programok megírásával kezdtem. Kezdetben iskolai beadandók során próbáltam ki magam, de rájöttem hogy a játékfejlesztés felé saját projektekkal lehet jobban elmélyedni. Ezért elkezdtem kisebb játékokat és interaktív szoftvereket fejleszteni, amivel nemcsak a programozás alapjait tudtam gyakorolni, hanem a játékok logikáját is elkezdtem jobban megérteni.

Komolyabban foglalkozni a programozással egyetem alatt kezdtem el. Ebben az időszakban rengeteg új dolgot tanultam, és megismerkedtem számos különböző programozási nyelvvel és technikával. Az egyetem nemcsak az elméleti tudást adta meg, hanem lehetőséget adott, hogy gyakorlati tapasztalatot szerezzek különböző feladatok és projektek révén.

A számomra eddig ismeretlen platformer játék stílussal is az egyetem alatt ismerkedtem meg. Az első találkozásom ezzel a műfajjal izgalmas kihívások elé állított amely lehetőséget adott arra, hogy új területeken is kipróbáljam magam a játékfejlesztésben. A platformerek különlegessége számomra az volt, hogy egyesíteni kell bennük a szórakoztató játékmechanikát, a kreatív pályadizájnt és az erőteljes történetmesélést. Az igazi áttörés az egyetemen megrendezésre kerülő GémDzsem alatt tapasztaltam. Olyan hatással volt rám ez az élmény hogy eldöntöttem: szeretnék a játékfejlesztéssel komolyan foglalkozni.

A szakdolgozatom során először készíték platformer játékot, amelynek animációit és grafikáit egy grafikus tervezi és készíti el. Ez új kihívást jelent számomra, mivel a játékmenet mellett most a vizuális elemekre is nagyobb figyelmet kell fordítanom, és együtt kell működnöm egy másik emberrel, hogy a játék mind vizuálisan, mind technikailag sikeres legyen. Az együtt működés lehetőséget ad arra hogy jobban megismerjem vizuális elemek kezelését, mivel eddig nem sokat dolgoztam grafikákkal.

Leadott forráskód elérhetősége: github.com/bartusjani/W5OLP9_szakdolgozat

Jelenlegi forráskód elérhetősége: [/github.com/bartusjani/Szakdoga_jelenegi_helyzet](https://github.com/bartusjani/Szakdoga_jelenegi_helyzet)

Játék elérhetősége: jankopepe.itch.io/diploma-game

Bemutató videó elérhetősége: youtu.be/SSNWiU138hQ

1. fejezet

Technológiai áttekintés

1.1. Godot Engine

A Godot Engine egy nyílt forráskódú és ingyenes játékmotor. Általános célú 2D és 3D játékmotor, ami mindenféle projektet támogat. Lehetővé teszi a játékok kiadását különböző platformokra. A Godotban lehet C#, C++ vagy GDScripttel programozni.

1.1.1. GDScript nyelv

A GDScript Godot specifikus nyelv. Ez a programozási nyelv egy objektumorientált, imperatív, magas szintű nyelv. A Pythonhoz hasonló behúzásalapú szintaxist használ. A Godot Enginehez optimalizált és integrált, célja hogy nagy rugalmasságot biztosítson a szoftverfejlesztéshez. A GDScript a Pythontól teljesen független, és nem arra épül.

A GDScript azonosítói kizárólag betűket (a-z, A-Z), számokat (0-9) és aláhúzás jelet tartalmazhatnak. Fontos hogy az azonosító nem kezdődhet számjeggyel. A nyelv kis és nagybetű érzékeny tehát például változo és a VALTOZO különböző változónak számítanak. Támogatja a UAX#31 szabványú Unicode karaktereket.

Az egész és lebegőpontos számokat aláhúzással (__) elválasztva is lehet írni. Például 123456789-et lehet 123_456_789-nek írni és a nyelv fel fogja ismerni.

A kommentet a Pythonhoz hasonlóan #-el lehet tenni. Lehet a kommentekből régiót csinálni ami össze csukható. A régiót így lehet csinálni #region ... #endregion [1].

Beépített adattípusok

Alapértelmezés szerint veremalapú objektumokként tárolódnak, amely érték szerint kerülnek átadásra.

Alapvető beépített típusok:

- null
- bool

- int
- float
- String

- StringName

Egy nem módosítható karakterlánc, ami biztosítja, hogy egy adott szöveg csak egyszer legyen a memóriában. Bár létrehozása erőforrás-igényesebb, gyorsabb összehasonlítást tesz lehetővé, ezért ideális szótárkulcsokhoz.

- NodePath

Egy előfeldolgozott útvonal csomópontokhoz, amely könnyen átalakítható String típusúvá.

Vektor adattípusok:

- Vektor2, Vektor2i

2D vektor típus ami x és y mezőt tartalmaz. A Vector2i-nél csak integer lehet az x és y mezőben.

- Vector3, Vector3i

3D vektor típus ahol x és y mező mellett van z mező is. Itt is csak integer lehet a Vector3i mezőiben.

- Transform2D

Egy 3×2 -es mátrix, ami 2D transzformációk végrehajtására alkalmas.

- Transform3D

Egy 3D transzformációt reprezentáló típus, amely egy Basis mezőből és egy Vector3 mezőből áll.

- Basis

Egy 3×3 -as mátrix amit 3D forgatás és skálázásra használnak.

1.1.2. A Godot verzióinak áttekintése

A Godot Engine-t folyamatosan fejlesztik, rendszeresen új funkciókkal, teljesítménybeli javításokkal és hibajavításokkal frissül. Az alábbiakban a legfontosabb verziókról fogok írni.

- Godot 1.0

2014 decemberében jelent meg a Godot Engine első stabil verziója. Több száz hibát javítottak és a közösség is jelentősen megnövekedett [2].

- Godot 2.0
2016 februárjában jelent meg. A Godot 2.0-ban javították a jelenetpéldányosítást. Bevezették a jelenet öröklést és egy új szöveges jelenetformátumot ami könnyebben kezelhető, Git kompatibilis és gyorsabb. Továbbá támogatja az onready kulcsszót és singletonokat [3].
- Godot 3.0
2018 januárjában jelent meg. A Godot 3.0-ban új fizikai alapú 3D renderelő kapott helyet. Behozták a GDNative-ot ami egy új keretrendszer amivel könnyen bővíthető a Godot C/C++ nyelven a motor újrafordítás nélkül [4].
- Godot 4.0
2023 márciusában jelent meg és jelentős javításokat hozott. A Godot 4.0-ban a 2D munkafolyamatokban új tilemap szerkesztőt vezettek be amivel könnyebb a szint tervezés. A 3D területén a shader-ek és VFX rendszerek újításokon estek át, emellett jelentős fejlesztéseket kapott és shader szerkesztő is [5].

Most a legfrissebb verziója a Godot 4.4.1 amit 2025 márciusában adtak ki.

1.1.3. Licenszi kérdések

A Godot az MIT licenc alatt készült és kerül kiterjesztésre. Az MIT licenc egyetlen követelménye, hogy a licenc szövegét valahol a játékban el kell helyezni [6].

1.1.4. A motor fő erősségei

- Intuitív jelenetvezérelt tervezés
A játékokat egyszerű blokkokból építheted fel, ahol a csomópontok (nodes) hierarchiája segít az átlátható kialakításában.
- Testre szabott kódolási eszközök
A GDScript és C# nyelvek biztosítanak gyors fejlesztést, míg a Godot 4.0 új statikus típusellenőrzése növeli a hatékonyságot és teljesítményt.
- Egyszerű, mégis nagy teljesítményű 3D motor
Támogatja a magas és az alacsony teljesítményű eszközöket, a Vulkan renderelő kiaknázza a játék GPU-k erejét.
- Speciális 2D munkafolyamat játékokhoz és alkalmazásokhoz
A dedikált 2D tile map editor lehetővé teszi a gyors világépítést, egyszerűsíti a logikát és a GUI rendszert a játékokhoz.

1.2. Unreal Engine

Az Unreal Engine az Epic Games által fejlesztett, nagy teljesítményű játékmotor. Az első verzióját 1998-ban adták ki. A legfrissebb verziója az Unreal Engine 5. A motor támogatja a C++ programozási nyelven való fejlesztést, de a Blueprint rendszere lehetővé teszi a vizuális programozást is. A motor teljesen ingyen használható egy bevételi határ alatt, így lehetőséget ad a kisebb fejlesztők számára is.

1.2.1. Blueprint

A Blueprint a játékmotor egyik fontos eszköze, amely lehetővé teszi a játékok, alkalmazások logikájának vizuális szkriptelését. Különösen hasznos a nem programozó felhasználóknak, mivel így mélyebb programozási tudás nélkül is képesek a projektjeiket megvalósítani.

A vizuális programozás erősségei

A Blueprint egy teljes játékmenet-szkriptrendszer, amely csomópont-alapú koncepción alapul. Objektumorientált osztályok vagy objektumok definiálására szolgál a motorban. Ez a rendszer rendkívül rugalmas és nagy teljesítményű, lehetővé teszi a tervezők számára, hogy az általában csak a programozók számára elérhető fogalmak és eszközök teljes skáláját használhassák.

Számos előnyt kínál a fejlesztés során. Lehetővé teszi az egyszerűbb definiálását a viselkedéseknek, ami megkönnyíti a szkriptelést és a projekt logikájának kialakítását. Ideális eszköz a prototípusok készítésére, mivel gyorsítja az osztályok létrehozását, módosítását, lefordítását és tesztelését, ezzel jelentős időt spórol meg. Az API-k gyorsabb felfedezését is lehetővé teszi [7],

Blueprint osztályok

A Blueprint osztály egy olyan eszköz, ami lehetővé teszi a új funkciók hozzáadását a játékmenethez kód írása nélkül. Eszközökként mentődnek el a tartalom csomagban [8].

- Data-Only Blueprint

Ez egy olyan osztály ami csak örökölt kódot, változókat, komponenseket tartalmaz. Lehetővé teszi ezek változtatását, de új elemeket nem lehet hozzáadni. Archetípusok helyettesítésére szolgál. Teljes Blueprintté alakítható ha kódot, változókat vagy komponenseket adunk hozzá.

- Level Blueprint

A Level Blueprint egy speciális Blueprint típus, amely a teljes szint globális ese-

ménygrafikájaként működik. Ez a Blueprint eseményeket, műveleteket kezel szintben lévő szereplőkkel kapcsolatban.

- **Blueprint Utilities**

A Blueprint Utility csak a szerkesztőre korlátozódik. Lehetővé teszi a szerkesztőben különböző műveletek végrehajtását vagy funkciók hozzáadását a szerkesztőben. Gyakran használják szkriptek, szerkesztőbeli kiegészítők létrehozására.

1.2.2. Unreal Engine verzióinak áttekintése

A Unreal Engine-t folyamatosan fejlesztik, rendszeresen új funkciókkal, teljesítménybeli javításokkal és hibajavításokkal frissül. Az alábbiakban pár újabb verzióról fogok írni.

- **Unreal Engine 4.27**

2021 augusztusában jelent meg. Ez a verzió mindenki számára kínál valamit kezdve a filmesektől, vizualizációs szakembereken át a játékfejlesztőkig. Ebben a kiadásban a munkafolyamatok egyszerűsítésén és a teljesítmény növelésén volt a hangsúly [9].

- **Unreal Engine 5.0**

2022 áprilisában jelent meg. Ezen kiadásban a lehetővé tették a fejlesztők számára hogy next-gen valós idejű 3D tartalmakat,élményeket hozzanak létre nagyobb szabadsággal, rugalmassággal és részletességgel. Az új funkciók mint a Nanite vagy a Lumen új vizuális minőséget biztosítanak, lehetővé téve a dinamikus világok létrehozását [10].

- **Unreal Engine 5.3**

2023 szeptemberében jelent meg. Ez a kiadás tovább javította a UE5 eszközöket. Fejlesztették a renderelést, világépítést, procedurális tartalomgenerálást, animációs és modellezési eszközöket, a szimulációkat. Ezáltal az Unreal Engine 5.3 kiadás tovább erősítette az Epic Games elkötelezettségét a valós idejű grafika és fejlesztői eszközök élvonalbeli fejlesztése iránt [11].

- **Unreal Engine 5.5**

2024 novemberében jelent meg. Ez legfrissebb verziója az Unreal Engine-nek. Jelentős előrelépések történtek az animációk készítésében, a virtuális produkcióban és a mobiljáték-fejlesztés területén is. Több funkció (például az in-camera-VFX, fejlesztői iteráció) elérte a gyártáskészséget [12].

1.2.3. Licenzi kérdések

Az Unreal Engine-t az Epic Games licencfeltételei alapján használható. Alapvetően ingyenesen elérhető, viszont egy meghatározott bevételi küszöb felett a fejlesztő köteles

fizetni a játékmotor használatáért. Az alábbi felsorolásban ismertetem a különböző licenc lehetőségeket [14].

- 1 millió dollár alatti bevétel esetén
Ingyenes a játék fejlesztők, egyéni fejlesztők, kisebb cégek, valamint iskolák és oktatók számára.
- 1 millió dollár feletti bevétel esetén
Kétféle fizetési lehetőség közül választhat a fejlesztő:
 - jogdíj alapú fizetés
Ha olyan játékot vagy alkalmazást készít a fejlesztő, amely futásidőben használja az Unreal Engine kódját, és harmadik fél számára kerül licencelésre, akkor kell jogdíjat fizetnie.
Költség: Az 1 millió dollárt meghaladó bevétel után 5% jogdíjat kell fizetni.
 - Ülőhely alapú fizetés
Ha az Unreal Engine-t kereskedelmi célokra használja a fejlesztő és az elmúlt 12 hónapban több mint 1 millió dollár bevételt termelt és nem olyan játékot vagy alkalmazást készít, amely futásidőben használja az engine kódját és harmadik félnek licencelhető, akkor ülőhely(seat) licenc díjat kell fizetnie.
Költség: Évi 777 549 Ft / ülőhely.

1.2.4. A motor fő erősségei

Az Unreal Engine számos erősséggel rendelkezik, amelyek hozzájárulnak ahhoz, hogy az egyik legnépszerűbb játékmotor legyen a piacon. Íme néhány főbb erőssége:

- Grafikai teljesítmény
Kiváló vizuális minőséget kínál, beleértve a valós idejű ray tracinget, amely valóság-hű fény- és árnyékhatásokat biztosít.
- Blueprint rendszer
A vizuális szkriptnyelv lehetővé teszi, hogy kódolás nélkül fejlesszünk játékokat, így gyors prototípusokat készíthetünk.
- Platform támogatás
Az Unreal Engine támogatja a legtöbb nagy platformot, beleértve a PC-t, konzolokat, mobil eszközöket és VR/AR eszközöket. Ez lehetővé teszi a fejlesztők számára, hogy széles körű közönséghez juttassák el alkotásaikat [13].

1.3. Unity

A Unity egy rendkívül népszerű és széles körben használt játékmotor. A Unityben lehetőség nyílik játékok, interaktív élmények és egyéb 3D- és 2D-alapú alkalmazások fejlesztésére. A fejlesztők különböző platformokon (például PC, konzolok, mobiltelefonok, VR/AR eszközök) készíthetnek a Unityvel alkalmazásokat.

1.3.1. Programozási nyelv

A Unityben a programozás alapértelmezetten C#-ban történik. A C# egy objektum-orientált programozási nyelv, amelyet a Microsoft fejleszt, és népszerű nyelv mivel könnyen tanulható és erőteljes eszközt biztosít a komplex játékmechanikák létrehozásához [15].

C# Főbb jellemzői

- Objektum-orientált
Lehetővé teszi a kódok szervezését osztályok és objektumok formájában, megkönnyítve ezzel a karbantartást és a bővítést.
- Garbage collection
A C# automatikusan kezeli a memóriát egy beépített szemétgyűjtő rendszer segítségével, ami csökkenti a memóriaszivárgás és a memória kezelése körüli hibák kockázatát.
- Kivételkezelés
A C# támogatja a kivételkezelést. Try-catch blokkokat alkalmaz, amely lehetővé teszi a hibák hatékony kezelését és a program hibamentes futását.
- Kompatibilitás a .NET és Mono környezetekkel
A C# kódot futtathatjuk a .NET keretrendszeren és az open-source Mono környezeten is, így lehetővé téve az alkalmazások futtatását különböző platformokon mint például Windows, Linux és macOS.

Fontos metódusok a Unityben

A Unity életciklusában a Start, Update és FixedUpdate metódusok kulcsszerepet játszanak a szkriptek működésében. A Start egyszer hajtódik végre, amikor az objektum aktívvá válik, míg az Update minden egyes képkockán fut, biztosítva a folyamatos működést, például a bemenetek feldolgozását. A FixedUpdate a fizikai számításokat végzi el, és rendszeresen hívódik meg az időközönkénti frissítésekkel, így ideális a fizikai interakciók kezelésére [39]. Létezik a LateUpdate is ami az összes Update után fut le.

Ez hasznos a szkriptek végrehajtásának sorrendjében. Ha a játékos kamerája követi a karaktert, és azt szeretnénk, hogy a kamera a karakter mozgása után álljon be a megfelelő pozícióra, akkor a LateUpdate tökéletes választás [40].

A fizikai események, mint az OnTriggerEnter és OnCollisionEnter, akkor aktiválódnak, amikor egy Collider objektum másik Colliderrel érintkezik, és lehetőséget biztosítanak az interakciók kezelésére, például a találatok vagy az objektumok közötti ütközések esetén [39].

1.3.2. Fontosabb verziók

A Unity folyamatosan fejlődő és frissülő játékmotor, amelyet az Unity Technologies fejlesztett ki. Az alábbiakban pár verziójáról fogok írni.

- Unity 4.0

2012 novemberében jelent meg. A Unity 4.0-nak a célja a meglévő technológia javítása és új funkciók bevezetése volt. Főbb újítások:

- Fejlett vizuális képességek minden platformra
Mobil platformokon is real-time árnyékok, skinned mesh instancing, normál térképek használata a fényképekhez.
- Mecanim animációs rendszer
Fejlett karakteranimációk létrehozására, beleértve az állapotgépeket, blend tree-eket, és auto retargetinget különböző karakterekhez.
- DirectX 11 támogatás
Teljes támogatás a shader model 5, tesszeláció és compute shaderek használatára a jobb grafikai teljesítmény érdekében [16].

- Unity 5.0

2015 márciusában jelent meg. Jelentős grafikai fejlesztésekkel és bővített szerkesztői funkciókkal jelent meg. Az új verzió lehetővé teszi a fejlesztők számára, hogy gyönyörű és innovatív játékokat készítsenek 21 platformra [17].

- Unity 6

2024 októberében jelent meg. Jelenleg ez a legfrissebb verziója a Unitynek. A verzió számos fejlesztést és újítást tartalmaz, ezek közül párat részletesebben is leírok:

- Grafikai teljesítmény javítás
A GPU Resident Drawer lehetővé teszi, hogy nagyobb és részletesebb világokat rendereljünk hatékonyan, miközben csökkenti a CPU költségeket. A GPU Occlusion Culling segítségével csak azokat az objektumokat rendereljük, amelyek láthatóak a képernyőn, így optimalizálva a teljesítményt.

Ezen kívül a Render Graph technológia, amely különösen mobilplatformokon hasznos, javítja a memória- és energiahatékonyságot, míg a PC-k és konzolok esetében magasabb szintű testre szabhatóságot biztosít.

- Több játékos fejlesztések

A Unity 6 erősíti a multiplayer fejlesztést a Multiplayer Center-el, amely egy központi hubként szolgál, segítve a multiplayer eszközök és szolgáltatások kiválasztását, anélkül, hogy bonyolult technológiai döntéseket kellene hozni.

- Több platformos fejlesztés

A Unity 6 egy új munkafolyamatot vezet be a Build Profile és Platform Browser funkciókkal, amelyek segítenek jobban kezelni és testreszabni a különböző platformokra készült build-eket.

- AI és vizuális fejlesztések

Az Adaptive Probe Volumes (APV) automatikusan elhelyezi a fénypróbákat, így gyorsabbá téve a fényalapú közvetett világításokkal való munkát. Az új Sentis AI lehetővé teszi a testreszabott AI modellek gyors integrálását [18].

1.3.3. Licenszelési kérdések

A Unity licenclési rendszer különböző típusú felhasználók és fejlesztők számára biztosít hozzáférést a motor különböző funkcióihoz. Az alábbi felsorolásban írok egyes licencekről.

- Unity Student

Ingyenes. kizárólag diákok vehetik igénybe.

- Unity Personal

Ingyenes. Bárki használhatja.

- Unity Pro

200\$ /hónap. Ezzel lehet konzolokra is kiadni.

- Unity Enterprise

A legnagyobb vállalatok számára kínált lehetőség. Ez a verzió a legfejlettebb eszközöket és prémium támogatást biztosít, és általában személyre szabott megoldásokat kínál [19].

1.3.4. Főbb erősségei

A Unity számos erősséggel rendelkezik, ebből párat részletesebben leírok.

- Egyszerűsített multiplayer játékfejlesztés

A Unity 6 integrált multiplayer platformot kínál, amely gyorsítja a játékok online funkcióinak fejlesztését és tesztelését, megkönnyítve a többjátékos élmények létrehozását.

- Fejlettebb mesterséges intelligencia képességek

Az új Sentis eszközkészlet lehetővé teszi a fejlesztők számára, hogy AI modelleket integráljanak a játékokba, dinamikus és interaktív élményeket teremtve ezzel a játékosok számára.

- Ray Tracing támogatás

Unity 6 natív támogatást ad a Ray Tracing API-hoz, lehetővé téve a fejlettebb és realisztikusabb fény- és árnyékolási effektusok alkalmazását, amik a grafikai minőséget jelentősen megemelik.

Ezek a fejlesztések a Unity 6-ban jelentősen növelik a játékfejlesztés hatékonyságát és minőségét, kielégítve a modern fejlesztők igényeit [18].

1.3.5. Jelenetek

A Unity-ben a fejlesztés jelenetekben (scenes) történik, amelyek a játék vagy alkalmazás különböző részeit reprezentálják. Minden jelenet önálló egységként működik, és különféle GameObjecteket, komponenseket, valamint beállításokat tartalmazhat, amelyek együtt alkotják az adott rész működését és megjelenését [30].

1.3.6. GameObject

Unityben a GameObject reprezentál mindent ami egy jeleneten belül lehet. A GameObject osztály sok metódussal rendelkezik amivel segíti a programozást, például GameObjectek közötti keresés, üzenetküldés, a GameObjectekhez kapcsolt komponensek hozzáadása vagy eltávolítása [21].

Komponensek

A komponensek minden GameObjectnek funkcionális darabjai. A komponensek szerkesztésével lehet a GameObjectek viselkedését módosítani. Komponenst lehet hozzáadni, szerkeszteni vagy törölni a GameObjectról. Egy GameObjectnek lehet több komponense is, viszont egy Transform komponenssel muszáj rendelkeznie, mivel ezzel lehet a GameObjectnek meghatározni a helyét, forgatását és a méretarányát [22].

Ebben a szekcióban bemutatom részletesebben azokat a komponenseket amiket használtam a szakdolgozatom fejlesztése során.

Transform

A Transform mint fentebb említettem a GameObject helyét, forgatását és méretarányát tárolja a jelenetben. A 2D térben a Transzformokat csak az x- vagy az y-tengelyen lehet módosítani. A 3D térben x-,y- és z-tengelyen lehet módosítani. A Unityben ezeket a tengelyeket a piros, a zöld és a kék színek jelölik. Ezeket lehet módosítani a jelenetben, a kódban vagy az inspector ablakban [23].

Collider

A Collider a Unity egyik legfontosabb alap komponense, amely lehetővé teszi hogy a GameObjectek fizikai kölcsönhatásba lépjenek egymással. A Collider egy fizikai határvonalat (ütköződobozt) határoz meg egy GameObject körül. Ez a határvonal láthatatlan a játékban, de a fizikai motor (Rigidbody) használja az ütközések, érintkezések, áthaladások stb. kezelésére. Collidert lehet 2D vagy 3D játékfejlesztésben is használni. A Collidereknek van isTrigger opciójuk ami ha be van állítva akkor nem ütköznek össze másik colliderekkel, de eseményeket tud kiváltani [24]. Sok különböző alakú Collider létezik például néhány fajtája:

- Circle Collider 2D/3D
- Box Collider 2D/3D
- Capsule Collider 2D/3D

RigidBody

A Rigidbody2D lehetővé teszi a fizikán alapuló viselkedést, például a gravitációt, tömeget lehet vele módosítani. Ez mozgatja a GameObjectet felülírva a Transform komponenst. Rigidbody komponens lehet használni 2D vagy 3D játékfejlesztésben is. Ha Collider komponens is van a GameObjecten akkor együtt mozog a Rigidbody-val [25].

Sprite renderer

A Sprite renderer komponensnek adnunk kell egy képet ami betölti és ez kontrollálja hogyan néz ki a jelenetben a GameObject Sprite-ja [26].

Light

A Light komponens segítségével fényforrásokat hozhatunk létre, amelyek megvilágítják a jelenet, árnyékokat vetnek. Light komponens lehet 2D vagy 3D játékfejlesztésben is [28]. Néhány fajtája:

- Pontfény
Minden irányba egyenletesen sugároz fényt.

- Irányított fény
Egyetlen irányba világít.
- Spot fény
Kúp alakú területben világít.

Script

A Unityben majdnem minden szkripttel vezérelhető. A GameObjectekhez kapcsolható Script komponensek a MonoBehaviour osztálytól öröklődnek. C# nyelvet használja. A szkriptekkel lehet befolyásolni a játék viselkedését. A szkriptek .cs fájlkként tárolódnak [27].

Animator

Az Animator komponenst akkor használjuk amikor a GameObjecthez valamilyen animációt szeretnénk társítani. Az Animator-nak viszont kell egy Animator Controller ami megmondja melyik animációkat használja [37].

Prefab

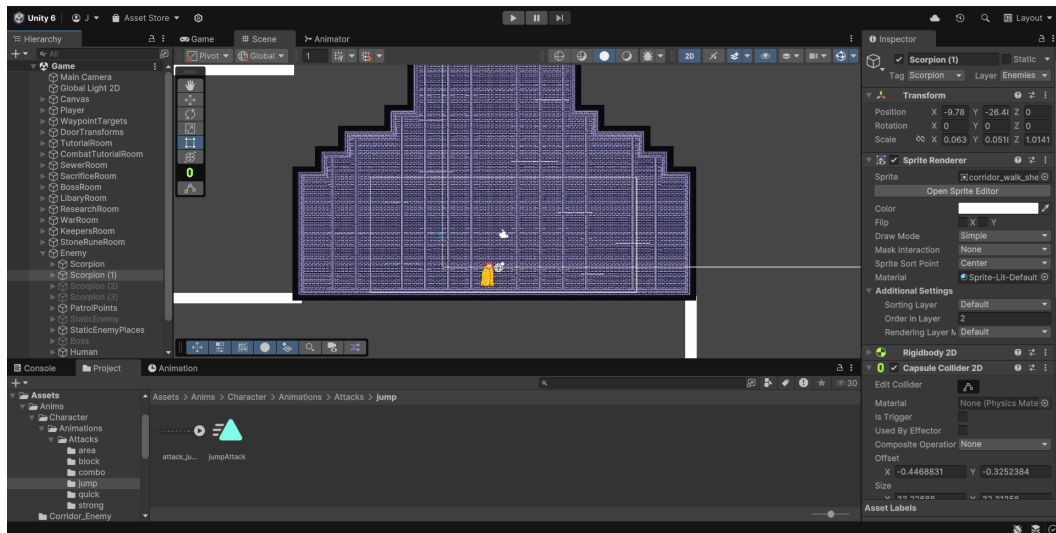
A Prefab az egy sablon. A Unityben prefabokban lehet tárolni kész GameObject-eket az összes komponensükkel, így könnyebben újrahasználhatók ezek. Különösen hasznos akkor amikor több példányra lenne szükség például egy ellenségből. Egy prefab használatával nem kell minden egyes alkalommal újra beállítani az objektum tulajdonságait vagy összerakni a komponenseit. Ehelyett egyszer elkészítjük és elmentjük a kívánt objektumot, majd ezt a sablont bármikor könnyedén drag & drop módszerrel hozzáadhatjuk bármelyik jelenethez, vagy akár szkripttel is létrehozhatjuk futás közben [29].

1.3.7. Editor

Az Editor egy grafikus felület ahol a fejlesztők összeállítják a jeleneteket, elhelyezik a GameObjecteket, beállítják a komponenseket és tesztelik a játékmenetet.

Scene View

A Scene View az Editor egyik legfontosabb része. Itt láthatod vizuálisan, szerkesztheted a játékot. A GameObjecteket itt választhatod ki, itt módosíthatod őket. A Scene View egy interaktív ablak, amely megkönnyíti a jelenetek felépítését. A jelenet nézete szabadon forgatható, nagyítható és mozgatható, így bármilyen szögből megvizsgálhatod az objektumokat. A különböző eszközök (például Move, Rotate, Scale) segítségével közvetlenül módosíthatod az objektumok elhelyezkedését a térben. Lehetőség van 2D



1.1. ábra. Unity Editor

és 3D nézet között váltani, attól függően, milyen típusú játékot fejlesztesz. A Scene View támogatja a rétegek és világítási módok megjelenítését is, ami segíti a jelenet pontos vizuális beállítását [30].

Game View

A Game View a játék végső, futtatott állapotát mutatja. Ez a nézet a projektben elhelyezett kamerák által renderelt képet jeleníti meg. A Game View gyakorlatilag azt mutatja, amit a játékos is lát majd játék közben. Ebben a nézetben tesztelhetők a játékmenet-elemek, például a mozgás, támadás vagy UI működése. A Game View támogatja a különböző felbontások és képarányok kipróbálását, így ellenőrizheted, hogyan viselkedik a játék többféle képernyőméreten. Beállíthatók simulated aspect ratio értékek, például telefonos, tabletes vagy monitoros nézetek is. Ezen kívül a Stat és Gizmos gombok segítségével megjeleníthetők technikai információk, például FPS, memóriahasználat vagy segédvonalak a hibakereséshez [31].

Inspector

Az Inspector ablakban található a kijelölt GameObject összes tulajdonsága és hozzárendelt komponense. Az Inspector dinamikusan frissül attól függően, hogy milyen objektum van kijelölve a Hierarchy vagy a Project ablakban. Itt lehet módosítani a GameObject pozícióját, forgatását, méretét (a Transform komponens segítségével), valamint egyéb hozzáadott komponensek beállításait is, mint például a Sprite Renderer, RigidBody2D, Collider, Script-ek, stb.

Az Inspector lehetővé teszi új komponensek hozzáadását a „Add Component” gomb segítségével, vagy meglévő komponensek eltávolítását és átrendezését. Továbbá, ha egy GameObject-hez egyedi C# script van csatolva, az Inspectoron keresztül meg lehet

adni annak publikus változóit, például referenciákat más GameObjectekre, sebességértékeket, animátor vezérlőket, és egyéb paramétereket, így nem szükséges a kódban módosítani ezeket.

Az Inspector ablak támogatja a Prefab objektumok kezelését is – lehetőség van Prefab példányok módosítására, valamint a változások alkalmazására vagy visszavonására az egész Prefabra nézve. Emellett az Inspector lehetőséget biztosít arra is, hogy az értékeket játék közben megfigyeljük és akár futásidőben módosítsuk őket, ami segít a tesztelésben és hibakeresésben is [33].

Hierarchy window

Ebben az ablakban minden GameObject megjelenik, amely a jelenetben található. Ezt az ablakot, amit Hierarchy ablaknak nevezünk, elsősorban a GameObjectek rendszerezésére és hierarchikus elrendezésére használjuk. Lehetőség van szülő-gyerek kapcsolatok létrehozására egyszerűen úgy, hogy egy objektumot ráhúzunk egy másikra – így a kapcsolt objektum együtt mozoghat vagy örökölheti a szülő tulajdonságait. A Hierarchy ablakban új GameObjecteket is létrehozhatunk a jobb gombos menü segítségével, például üres objektumot, 2D sprite-ot, kamerát vagy UI elemet.

A Hierarchy ablak segítségével gyorsan kiválaszthatjuk az objektumokat a szerkesztéshez, illetve átnevezhetjük őket a jobb áttekinthetőség érdekében. A jól szervezett Hierarchy jelentősen megkönnyíti a projektek kezelését, különösen akkor, ha több száz objektum van jelenetben. A Hierarchy a jelenet vizuális felépítésének tükrözésére szolgál, és együttműködik az Inspector és a Scene ablakokkal [33].

Project window

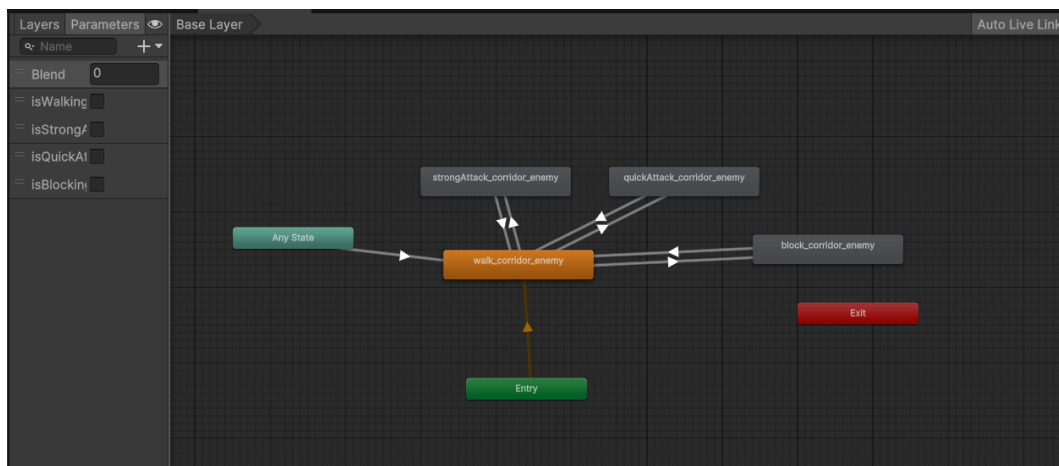
Ebben az ablakban megjelenik minden fájl/mappa ami a projektben van, itt tudsz navigálni a fájlok között [35].

Animator window

Az Animator lehetővé teszi az animációk és azok közötti átmenetek kezelését és irányítását. Általában több animációra van szükség és ezek között átmeneteket kell létrehozni amikor bizonyos játékbeli események történnek. Például ha a játékos megnyomja az SHIFT gombot akkor séta animációból átmegy futás animációba. Az átmeneteket állapot géppel kezeli az Animator [32].

Animation window

Az Animation ablakban lehet megnézni, módosítani, hozzáadni az animációkat a GameObjecteken. A GameObjecten kell lennie egy Animator komponensnek hogy a GameObject animáció látszódjon a játékban [36].



1.2. ábra. Unity Animator

2. fejezet

Rendszerterv

2.1. A rendszer célja

A szakdolgozatom célja egy kétdimenziós platformer típusú videojáték fejlesztése, amelyben a játékos egy oldalsó nézetű világban irányít egy karaktert. A játék során a játékos különböző pályákon halad keresztül, miközben akadályokat küzd le, ugrásokkal manőverezik, valamint különböző típusú ellenségekkel harcol. A pályák teljesítésével a játékos fokozatosan halad előre a szinteken, egyre nehezebb kihívásokkal szembesülve. A fejlesztés során a grafikai elemeket – beleértve háttereket, karakter animációkat, felhasználói felület elemeket valamint az összes dizájnhoz kapcsolódó komponenst – egy külön grafikus készítette. A grafikus szakdolgozati munkája szorosan kapcsolódik az én projektemhez, ugyanis ő saját szakdolgozata keretében dolgozta ki és valósította meg a játék látványvilágát. Ez az együttműködés lehetővé tette, hogy a játék egységes vizuális stílusban készüljön el, amely nemcsak esztétikai szempontból emeli a színvonalat, hanem hozzájárul a játék hangulatának és egyediségének kialakításához is. A közös munka során a látványterveket és az animációkat folyamatosan egyeztettük, hogy azok minél jobban illeszkedjenek a játékmenethez és a játék által képviselt világ hangulatához.

2.2. Projektterv

A projekt célja egy kétdimenziós platformer videojáték elkészítése Unity játékmotor segítségével, amelyben a játékos harcol, ugrál, és különféle akadályokat leküzdve jut tovább a szinteken. A projekt megvalósításához két fő erőforrás állt rendelkezésre: a fejlesztő (jelen szakdolgozat készítője) és a grafikus, akik külön-külön, de összehangoltan dolgoztak a játék létrehozásán.

Szerepkörök

- Fejlesztő

A játék mechanikáinak, működésének, szintlogikájának és programkódjának megvalósítása. Felelős volt a Unity projekthez kapcsolódó összes technikai döntésért, valamint a grafikus által készített assetek integrálásáért.

- Grafikus

A játékhoz szükséges összes vizuális elem – háttérképek, karakterek, animációk, UI-elemek – megtervezése és elkészítése. A grafikus saját szakdolgozata keretében dolgozott, így a vizuális tartalom elkészítése önálló projektként, de szorosan együttműködve zajlott.

Felelősségek

Név	Szerep	Felelősségek
Szakdolgozó	Fejlesztő	Játékmenet logika, programozás és hibajavítás, amelyek során a játékmenet mechanikáit fejleszttem, és a felmerülő hibákat gyorsan orvoslom.
Grafikus	Látványtervező	Sprite-ok, animációk, UI elemek és háttérképek elkészítése, amelyek segítik a vizuális élmény megteremtését és a játék világának kifejeződését.

A fejlesztő és a grafikus rendszeres konzultációk során hangolták össze a játékmenetet és a látványvilágot, hogy a kész játék esztétikailag is egységes élményt nyújtson.

Mérföldkövek

- **Alap játékmechanikák elkészítve**

A játékos mozgása, ugrása és az ütközésérzékelés már működőképes. A karakter irányítható, képes a pályaelemekre reagálni, és a mozgásmechanikák stabil alapot adnak a játékmenethez.

- **Ellenségek implementálva**

Az összes ellenség működik, képes felismerni a játékost, üldözni, illetve támadást kezdeményezni. Az alap harcrendszer már működik.

- **Első játszható szint elkészítve**

Elkészült egy teljes pálya, amely bejárható a játékos által. Bár a vizuális elemek még hiányoznak, a szint szerkezete és funkcionalitása már tesztelhető.

- **Grafikai integráció befejezve**

A grafikus által készített sprite-ok és animációk beépítésre kerültek. A karakterek, ellenségek és környezeti elemek már vizuálisan is megjelennek a játékban.

– **Teljes játszható verzió készen**

A játék több szinttel, működő harcrendszerrel és vizuális tartalmakkal játszható állapotban van. Az alap játékmenet teljes egészében kipróbálható.

– **Tesztelés és finomhangolás**

A játék stabilan fut, a főbb hibák javítva lettek. A mechanikák finomhangolásra kerültek, a játékmenet kiegyensúlyozott és gördülékeny.

2.3. Funkcionális terv

A funkcionális terv célja a játék funkcióinak részletes technikai leírása a fejlesztő szemszögéből, a funkcionális specifikáció alapján. A terv bemutatja a rendszer szereplőit, a fő használati eseteket és azok lefutását, a határosztályokat, valamint a menürendszer és a képernyők logikai felépítését. A játék Unity játékmotorban készült, C# programozási nyelven.

2.3.1. Rendszer szereplők

Szereplő	Leírás
Játékos	A játék fő irányítója, aki a karakter mozgását, támadásait és interakcióit végzi, miközben különböző akadályokat és kihívásokat old meg a játék során.
Ellenség	AI által vezérelt ellenfél, amely a játékost támadja, vagy akadályozza, reagálva a játékos mozgására és stratégiájára, hogy fokozza a játék kihívását.
Játék-rendszer	A háttérben működő logika, amely kezeli a szintek váltását, a karakterek életerejét, pontszámokat és egyéb alapvető játékelemeket, biztosítva a zökkenőmentes és dinamikus játékmenetet.

2.1. táblázat. A játék szereplői és funkcióik

2.3.2. Rendszerhasználati esetek és lefutásaik

1. Játék indítása

Használati eset: A játékos elindítja a játékot a főmenüből.

Lefutás:

- A főmenü megjelenik.
- A játékos a „Start” gombra kattint.
- A rendszer betölti az első pályát.

2. Mozgás és ugrás

Használati eset: A játékos a karakterrel mozog és ugrik.

Lefutás:

- A játékos lenyomja a bal/jobbs nyílbillentyűt vagy „A” / „D” billentyűt.
- A karakter ennek megfelelően mozog balra vagy jobbra.
- A játékos megnyomja a „W” billentyűt, a karakter ugrik.

3. Harc ellenséggel

Használati eset: A játékos közelharcban vagy távolsági támadással ellenséget támad.

Lefutás:

- A játékos lenyomja a támadás gombot.
- A karakter animációja lefut, a közelben lévő ellenség életerejé csökken.
- Az ellenség visszatámadhat.
- Ha az életerejé 0, az ellenség meghal.

4. Szoba teljesítése

Használati eset: A játékos teljesíti a szobát.

Lefutás:

- A játékos belép a szoba végét jelző trigger zónába.
- A rendszer betölti a következő szobát.
- A játékmenet folytatódik.

2.3.3. Menühierarchiák

– Főmenü

- Játék indítása
- Beállítások
 - Hang beállítások
 - Képernyő felbontás
 - Teljes képernyő
- Kilépés

– PauseMenü

- Folytatás

- Beállítások
 - Hang beállítások
 - Képernyő felbontás
 - Teljes képernyő
- Kilépés

2.4. Követelmények

A követelmények a rendszer fejlesztése során követendő irányelvek és célok, amelyek a játék funkcionalitását és működését határozzák meg. A követelmények a fejlesztési fázis során változhatnak, de az alábbiakban szereplő követelmények képezik a rendszerterv alapját. Ezek a követelmények az elkészítendő játék megvalósítását célozzák meg, és minden fontos funkciót, teljesítendő elvárást tartalmaznak.

2.4.1. Funkcionális követelmények

- Felhasználói felület

A játék képernyőjének bal felső sarkában jelenik meg a játékos életereje és állóképessége, egy könnyen olvasható, csík formájában. A jobb felső sarokban az objektívet és információt tartalmazó üzenetek jelennek meg, a képernyő alján megjelenelő szövegbuborékokban lehet olvasni a karakter dialógusait, ezek a játék menete alapján frissülnek és új információkat adnak.

- **Elvárás:** A UI-elemek skálázható méretűek és jól látható betűtípust használnak. A fontos információk mindig láthatóak, és nem takarják ki a játéktér fontos részeit.

- Tárgyak kezelése

A pályákon elhelyezett tárgyak nem befolyásolják közvetlenül a játék előrehaladását, azonban háttértörténeti információkat tartalmaznak. A játékos az I betű lenyomása után megtekintheti a tárgyak listáját, a képernyő közepén megjelenő ablakban.

- **Elvárás:** Az inventory-ban a tárgyra való kattintás után a tárgyak listája mellett megjelenik a tárgy neve és leírása. Az UI a Canvas rendszerre épül, és a tárgylista Scroll View komponenssel görgethető.

- Harc rendszer

A játékos kizárólag közelharc támadásokat hajthat végre, amelyeket kijelölt bilentyűk lenyomásával aktiválhat. A támadásokhoz különféle animációk társulnak

(pl. alaptámadás, ugrótámadás, erős ütés), amelyek az Animator komponens segítségével váltakoznak a játékos aktuális állapota szerint (pl. talajon vagy levegőben van-e).

Az ellenségek többsége szintén közelharcra képes, és meghatározott támadási mintát követ: például a játékos közelébe érve visszatámadnak, vagy egy rövid töltési idő után gyors támadást hajtanak végre. Az egyik speciális ellenség képes távolsági támadásra, lövedék kilövésére, így a játékosnak nagyobb figyelmet kell fordítania a pozícionálásra.

- **Elvárás:** A közelharc rendszer gyors és reszponzív, lehetővé téve a játékos számára, hogy kombókat hajtson végre (például egymás utáni támadásokkal) és speciális mozdulatokat alkalmazzon ugrás vagy futás közben. A támadásokhoz ütközésérzékelés (BoxCollider2D, Raycast stb.) társul, amely pontosan meghatározza, hogy a támadás eltalálta-e az ellenséget.
- **Játékos mozgás**

A karakter mozgása az A, D betűk lenyomásával történik alapvetően, de a bal és jobb irányú nyílbillentyűkkel is lehet irányítani, míg az ugrás a „W” vagy a fel nyíl lenyomásával hajtható végre. A mozgás RigidBody2D komponenssel történik, a súrlódást és gravitációt figyelembe véve.

- **Elvárás:** A karakter mozgása folyamatos és dinamikus, amit a FixedUpdate() metódusban történő sebességfrissítés garantál. Az animációk (Animator komponenssel) automatikusan reagálnak a mozgás sebességére és irányára (xVelocity, yVelocity változók alapján). Az ugrás csak akkor aktiválódik, ha a karakter talajon van, amit a OverlapCircle segítségével ellenőriz a rendszer. Az ugrás ereje egy fix jumpingPower értéken alapul.
- A karakter irányt vált, ha ellenkező irányba kezd mozogni, amit a Flip() függvény valósít meg a transform.localScale.x tükrözésével.

2.4.2. Nem funkcionális követelmények

- **Teljesítmény**

A játék futása simának kell lennie minden célszámítógépen, legalább 30 FPS (képkocka/másodperc) sebességgel.

 - **Elvárás:** A játéknak gyorsan reagálnia kell, és nem szabad akadoznia még nagyobb szintű grafikai elemek vagy több ellenség jelenléte esetén sem.
- **Hozzáférhetőség**

A játék kezelőfelületének könnyen használhatónak kell lennie minden játékos számára.

- **Elvárás:** felhasználói felület egyszerű, könnyen navigálható és átlátható legyen.
- Kép- és hangminőség
A grafikáknak és a hangoknak élesnek, tisztának kell lenniük, és nem szabad akadozniuk, illetve elcsúszniuk a játékos mozgásával.
- **Elvárás:** A grafikai elemeknek és hangoknak összhangban kell lenniük a játékmenettel és a játék hangulatával.

Ezek a követelmények képezik a fejlesztés alapját, és a jövőbeni fejlesztési fázisok során figyelembe kell venni őket a megfelelő tervezési és implementálási döntések meghozatalakor.

2.5. Használt fejlesztői eszközök

2.5.1. Unity

A játék fejlesztése a Unity játékmotor segítségével történt. A Unity a világ egyik legnépszerűbb fejlesztői környezete, amely kiemelkedő rugalmasságot biztosít a különböző típusú játékok készítésében, legyen szó 2D vagy 3D játékokról. A játékfejlesztési folyamat során a Unity biztosította az összes szükséges eszközt a játékmenet logikájának, a grafikai elemek kezelésének, az animációk és fizikai rendszerek integrálásának, valamint a felhasználói interakciók kezelésének megvalósításához. A Unity fejlesztői környezetet és komponenseit a 1.3. szekcióban mutatom be.

2.5.2. Github, Git

A GitHub egy felhőalapú verziókezelő platform, amely a Git verziókezelő rendszert használva lehetővé teszi a projektek tárolását, megosztását és a fejlesztők közötti együttműködést. A Git lehetővé teszi a fájlok különböző verzióinak kezelését, míg a GitHub biztosítja a központi tárolót, ahol a fejlesztők szinkronizálhatják munkájukat [41].

2.5.3. Trello

A Trello egy vizuális projektmenedzsment eszköz, amely segít a feladatok és projektek rendszerezésében. Kártyák és táblák segítségével a csapatok könnyen nyomon követhetik a munka előrehaladását. A felhasználók különböző kategóriákba rendezhetik feladataikat, hozzárendelhetnek határidőket, és megoszthatják a munkát másokkal [42].

3. fejezet

Saját projekt fejlesztése

3.1. A platformer játék bemutatása

A szakdolgozat keretében egy 2D-s platformer játékot fejlesztettem Unity játékmotor segítségével. A játék lényege, hogy a játékos különböző szinteken, úgynevezett „szobákban” halad keresztül, miközben különféle ellenségekkel harcol, akadályokat kerül ki, és a célpont (waypoint) irányába mozog. A játékos számára a fő kihívást az ellenségek legyőzése, illetve a szintek közötti előrehaladás során szükséges platformugrások jelentik.

A játék vizuális elemeit – háttér, karakterek, animációk, UI – egy külön grafikus tervezte és készítette, aki szintén a szakdolgozatát ebben a témában írta. A látványvilág egyedi, pixel art stílusú, így kifejezetten karakteres megjelenést biztosít a játéknak.

3.1.1. A játékos karakter

A játékos egy karaktert irányít, aki képes:

- balra és jobbra mozogni
- ugrani
- különféle támadásokat végrehajtani mint például a gyors, erős területi, ugró támadás.

A karakter rendelkezik életerővel, amely csökken, ha az ellenségek megsebesítik. Ha az életerő eléri a nullát, a karakter meghal, és a játék az utolsó checkpoint-tól indul újra. A karakter rendelkezik még állóképességgel is, amely csökken, ha fut vagy erős, területi támadást használ. A karakter mozgása és animációi pontosan követik a játékos parancsait, ezzel biztosítva a gördülékeny játékmenetet.

3.1.2. Ellenségek

A játékban többféle ellenség jelenik meg, amelyek különböző támadási és mozgási mintákkal rendelkeznek. Ezeket hardcode-olt AI vezérli, és a játékos közelébe érve aktívvá válnak.

– Skorpió

- **Mozgás:** Alapvetően, ha a játékos nincs az ellenség üldözési távolságában, az ellenség két kijelölt pont között oda-vissza mozog.
- **Viselkedés:** Amikor a játékos belép az ellenség üldözési távolságába, a skorpió elindul felé, és ha elég közel ér hozzá, támadásba lendül és elkezd támadni a játékost.
- **Támadásai:** A skorpió támadásai közelharciak:
 - Gyors támadás: A gyors támadás egy gyors, alacsonyabb sebzéssel járó támadás, amely lehetőséget ad az ellenség számára, hogy gyorsan reagáljon a játékos mozgására. A gyors támadások kisebb sebzést okoznak, de sokkal gyorsabban végrehajthatók, lehetővé téve az ellenség számára, hogy folyamatos nyomást gyakoroljon a játékosra.
 - Erős támadás: Az erős támadás egy lassabb, de erősebb támadás, amely nagyobb sebzést okoz a játékosnak. Az ellenség a támadás során jobban felkészül, hogy egy erőteljes ütéssel sújtsa a játékost. Az erős támadás lassabb mint a gyors, így a játékosnak van lehetősége elkerülni vagy blokkolni, de sikeres eltalálás esetén komolyabb sebzést eredményez.

Ezen kívül a skorpió képes blokkolni a játékos támadásait, miközben a saját támadásai között váltogat.

– Csatornai Ellenség

- **Mozgás:** Alapvetően ez az ellenség statikus, egy helyen áll, és csak akkor mozdul el, ha az életerejé 20 alá csökken.
- **Viselkedés:** Az ellenség viselkedése a saját életerejétől és a játékos helyzetétől függ:
 - Ha a játékos a szobában tartózkodik, az ellenség folyamatosan támadja őt.
 - Ha a játékos 3 lövedékét blokkolja, utána egy ideig nem tud támadni. Így a játékos egy rövid szünetet kap, hogy reagálhasson.
 - Amikor a saját életerejé 20 alá csökken, akkor elmozdul a helyéről, hogy új pozíciót vegyen fel. Ilyenkor az életerejé vissza megy a max életerejére. Ezt a műveletet legfeljebb kétszer ismételheti meg.

◦ Miután kétszer elmozdult, a játékos szabadon legyőzheti.

• **Támadásai:** A támadásai távol harciak:

- Egy lövedéket lő ki a játékos felé. Ha ez a lövedék egy akadállynak ütközik, például egy falnak vagy a játékos pajzsának, akkor a lövedék megsemmisül, és nem okoz sebzést.
- Ha a lövedék eléri a játékost, akkor sebzést okoz.

– Ember

- **Mozgás:** Amikor a játékos nem tartózkodik a szobában, az ellenség inaktívvá válik, és csak egy idle animációt hajt végre. Ebben az állapotban nem reagál semmilyen eseményre, és nem végez támadásokat vagy egyéb akciókat, amíg a játékos vissza nem tér a szobába.
- **Viselkedés:** Amikor a játékos a szobában tartózkodik, az ellenség automatikusan elindul felé. Amint elég közel kerül a játékoshoz, megkezd a támadást. Ez a viselkedés biztosítja, hogy az ellenség mindig aktívan reagáljon a játékos jelenlétére, és ne maradjon inaktív, ha a játékos a közelben van.
- **Támadásai:** Az ember támadásai közelharciak:
 - gyors támadás
 - Erős támadás
 - Roham támadás: Amikor a játékos belép az ellenség rohamkörébe, az ellenség aktiválhatja a roham támadást, amely során hirtelen nekiront a játékosnak, és gyorsan elér egy közvetlen támadást. Ez a mozdulat lehetővé teszi az ellenség számára, hogy gyorsan reagáljon a játékos jelenlétére.

Ezen kívül az ember ellenség képes blokkolni a játékos támadásait, miközben a saját támadásai között váltogat.

– Boss

- **Mozgás:** Amikor a játékos nem tartózkodik a szobában, az ellenség inaktívvá válik, és csak egy idle animációt hajt végre. Ebben az állapotban nem reagál semmilyen eseményre, és nem végez támadásokat vagy egyéb akciókat, amíg a játékos vissza nem tér a szobába.
- **Viselkedés** Amikor a játékos a szobában tartózkodik, az ellenség automatikusan elindul felé. Amint elég közel kerül a játékoshoz, megkezd a támadást. Ez a viselkedés biztosítja, hogy az ellenség mindig aktívan reagáljon a játékos jelenlétére, és ne maradjon inaktív, ha a játékos a közelben van.

- **Támadásai:** A támadásai neki is közelharciak, viszont az életerejétől függően 2 fázisra vannak osztva a támadások.
 - Első fázis: Amikor az életereje 150 és 300 azaz a max életereje között van:
 - * Erős támadás
 - * Területi támadás: A területi támadás olyan támadást jelent, amely egy nagy területen ad sebzést. Az ellenség egy szélesebb hatótávolságú támadást hajt végre, és ha a játékos benne van ebben a hatótávban akkor sebzést kap.
 - Második fázis:
 - * Roham támadás
 - * Gyors támadás
 - fázistól függetlenül tudja a játékos támadásait kivédeni, miközben a saját támadásai között váltogat.

3.1.3. Pályarészek közötti átjárás

A pályá úgy van kialakítva, hogy külön „szobákra” van osztva. Minden szoba egy különálló rész a játéktérben, és a játékos egy átjárón keresztül tud átjutni a következőbe.

A szobák közti átjárás logikája:

- A játékos eléri a kijáratot a szobában.
- Egy „trigger” érzékeli a belépést. Itt két dolog történhet:
 - Ha a játékos teljesítette a szobában lévő feladatot akkor át engedheti
 - Ha még nem teljesítette akkor viszont ki ír a képernyőre egy üzenetet, ami tartalmazza a feladatot.
- A rendszer átrakja a karaktert a következő szobába miközben egy rövid ideig elsötétül a képernyő.
- A játékos pozíciója automatikusan beállításra kerül az új szobán belüli kezdőpontra.

Ha a játékos szeretné vissza is tud menni egy előző szobába és onnan már egyből tud tovább menni a következő szobába.

3.1.4. Waypoint mutató

A játékos tájékozódását egy waypoint mutató segíti, amely mindig az aktuális cél irányába mutat:

- Ha a cél (pl. következő kijárat, fontos tárgy, vagy ellenség) nem látszik a képernyőn, a waypoint ikon a képernyő szélén jelenik meg, és az irányt mutatja.
- Amennyiben a játékos a szoba aktuális célpontjának közelébe kerül, a mutató automatikusan eltűnik. A cél teljesítése után a mutató ismét megjelenik, és a következő cél irányába kezd el jelezni.
- Ezzel a megoldással a játékos nem téved el, és a játék élménye áramvonalasabbá válik.

Ezzel a modullal biztosítottuk, hogy a játékos ne akadhasson el a pályán, lehetővé téve a zökkenőmentes előrehaladást.

3.2. Játékmechanikák és algoritmusok

3.2.1. Mozgás és irányítás

A játékos mozgása a 2D-s fizikai motor segítségével lett implementálva, amely biztosítja a gravitációs hatásokat, valamint az ütközéseket és a platformok közötti interakciókat. A játékos irányítása az Input System-en keresztül történik, amely lehetővé teszi a billentyűzet és az egér használatát.

Ellenségek Mozgása

A játékban alkalmazott ellenségek AI-ja hardcode-olt (kézzel kódolt), amely az ellenség viselkedését és reakcióit előre meghatározott logikák alapján irányítja. Az AI működését statikus algoritmusok és feltételek szabályozzák, amelyek a játékos jelenlétére és a játékhelyzetekre reagálnak.

Csatornai ellenség

A csatornai ellenségnek az AI-ja csak akkor mozgatja az ellenséget ha az életerejé 20 alá csökken, mint egy védelemi mechanizmusként. Itt a kódban meghatározott helyek tömbjében mozog, ha eléri a tömb végét akkor már nem tud helyet váltani és ekkor lehet legyőzni.

Skorpió ellenség

Az ellenség mozgása előre meghatározott pályákon zajlik. A mozgás logikája a játéktér egyes meghatározott pontjaira (waypoint) irányítja az ellenséget, amelyeket a fejlesztés során statikusan kódoltam. Az AI folyamatosan ezen pontok között mozog, és egy egyszerű irányítást követ, amely nem alkalmaz semmiféle dinamikus döntéshozatalt vagy alkalmazkodást. A skorpió AI figyeli, hogy van-e alatta talaj, hogy biztosan ne essen le a pályáról. Ha nem érzékel talajt, a skorpió automatikusan megfordul, és a másik irányban lévő waypoint-hoz kezd el mozogni. Ez a viselkedés segít abban, hogy az ellenség ne menjen le a szoba legaljára amikor a játékos a szobában tartózkodik, és folyamatosan a kijelölt helyén maradjon, miközben a játékost várja. Minden másik ellenség csak akkor kezdi el a mozgást ha a játékos beért az üldözési távolságába.

3.2.2. Támadások

A játékban az ellenség és a játékos támadásainak sebzésrendszere a következő módon van megvalósítva: minden támadásnak van egy meghatározott pontja a játékos és az ellenség előtt, amely körül a támadás sebzést ad. Ez a kör a támadások hatókörét reprezentálja, és minden támadás során az adott körben lévő célpontok, például a játékos, sebzést kapnak. A támadások hatóköre és a sebzés nagysága az egyes támadások típusaitól függően változhat, és a játékos vagy az ellenség helyzete alapján kerül kiszámításra.

Kivételt képez azonban a csatornai ellenség támadása, mivel az ő támadása nem a megszokott módon működik. A csatornai ellenség egy lövedéket instanciál a támadási pontján, amely a játékos felé halad. A lövedék fizikai tulajdonságait és működését a hozzá tartozó Collider komponens irányítja. A lövedék esetében a sebzés akkor kerül kiosztásra, ha a lövedék eltalálja a játékost, és a Collider ütközik a játékos Colliderével. A lövedék akkor nem okoz sebzést, ha valamilyen akadályba ütközik, vagy ha a játékos blokkolja azt, ekkor a lövedéket töröli a rendszer, hogy az ne okozzon további problémát a játékmenetben.

3.2.3. Pályarészek közti átjárás logikája

A játékban, amikor a játékos eléri az ajtót, a rendszer automatikusan ellenőrzi, hogy a játékos teljesítette-e a szoba célját. Ha a cél nem teljesült, a rendszer egy üzenetet jelenít meg, amely tartalmazza a szoba célját, tájékoztatva a játékost arról, hogy még nem érte el a szükséges feltételeket a továbbjutáshoz.

Abban az esetben, ha a játékos sikeresen teljesítette a szoba célját, a rendszer egy „Inspect” feliratot jelenít meg az ajtó mellett, jelezve, hogy a játékos interakcióba léphet az ajtóval. Ekkor, ha a játékos az E billentyűt lenyomja, a rendszer átvált a következő

szobára, lehetővé téve a továbbhaladást.

A játékban az ajtók előtt Collider2D komponensek találhatók, amelyek trigger-re vannak állítva. Ez azt jelenti, hogy az ajtók területére lépve a rendszer automatikusan érzékeli a játékos jelenlétét. Az ellenőrzés csak akkor kezdődik el, amikor a játékos belemegy a trigger zónába. A rendszer a játékos címkéjét (tag) keresi a GameObject-en, hogy megbizonyosodjon arról, hogy valóban a játékos lépett a területre, és ne aktiválódjon más objektumoknál a trigger. Ez a megoldás biztosítja, hogy az ajtó előtt található collider csak akkor aktiválódjon, ha valóban a játékos lép be a területre, így elkerülhető, hogy más objektumok, például ellenségek vagy statikus elemek is véletlenül aktiválják a továbbjutási mechanizmust.

Waypoint mutató

A waypoint a játékban egy kis ikon, amely a Canvas-ban jelenik meg, és a kód irányítja, hogy éppen melyik irányba mutasson. Amikor a játékos elég közel ér a célhoz, a waypoint automatikusan eltűnik. Az irányítás és a célpontok közötti váltás a `setWaypointTarget` metódus segítségével történik, amelyet a kódban használtam. Ezt a metódust alkalmazom a Collider trigger enter eseményeknél, valamint a kontroller kódokban, hogy dinamikus frissítsem a waypoint célpontját, amikor a játékos elér egy új szobát vagy új célt.

4. fejezet

Tesztelés

A tesztelés a szoftverfejlesztés egyik legfontosabb és elengedhetetlen lépése, amely lehetővé teszi a hibák korai felismerését és kijavítását. A tesztelés során különböző módszerekkel próbáltam reprodukálni a felhasználói interakciókat, ellenőrizve, hogy a rendszer megfelelően reagál-e minden lehetséges helyzetben. A hibák észlelése és javítása nélkülözhetetlen, mivel bármilyen figyelmen kívül hagyott probléma komoly hátrányt okozhat a játék működésében, élményében és teljesítményében.

Én két féle tesztelési módot alkalmaztam a projektben: Playtest-et ahol körültekintően kipróbáltam minden lehetséges dolgot, itt barátaim is besegítettek a hibák keresésében. Alkalmaztam a Unity-n belül Unit testeket amit a unity Test Framework moduljával csináltam.

4.1. Playtest

A játék fejlesztése során már a korai fázisokban megkezdtem a play tesztelést, hogy biztosítsam a funkcionalitás és a játékelmény folyamatos fejlődését. A tesztelés során különös figyelmet fordítottam arra, hogy minden új funkció alapos kipróbáláson menjen keresztül. Ez lehetőséget adott arra, hogy a fejlesztési ciklus minden szakaszában azonosítani tudjam a hibákat és problémákat, valamint azokat a lehető leghamarabb kijavítsam. Az azonnali hibajavításra való törekvés biztosította, hogy a játékelmény ne szenvedjen csorbát, és hogy a felhasználói élmény ne romoljon a fejlesztési időszak alatt.

Amikor a játék fejlesztése már olyan szintre ért, hogy több ellenség és szoba is bekerült a játékba, úgy döntöttem, hogy a barátaimnak is elküldöm az akkori verziót kipróbálásra. Ez a lépés lehetőséget biztosított arra, hogy külső szemlélők is visszajelzéseket adjanak, és így biztos lehettem abban, hogy esetleg olyan hibákat, amelyek felett én átsiklottam, ők könnyen észrevesznek. A barátaim nemcsak logikai hibákat találtak, hanem hasznos javaslatokkal is segítettek. Például tippeket adtak a karakterek animációival kapcsolatban, hogy hogyan lehetne azokat gyorsabbá vagy lassabbá tenni,

hogy jobban illeszkedjenek a játékmenet dinamikájához és a felhasználói élményhez. Az ő visszajelzéseik rendkívül hasznosak voltak, mivel lehetővé tették, hogy finomhangoljam a játék különböző aspektusait, és a játékelmény még élvezetesebbé váljon. Ezen kívül az ő friss szemszögük segített a játékmenet újabb rétegeinek kidolgozásában is, amelyek a fejlesztés következő szakaszaiban komoly előnyökkel jártak.

4.2. Unit teszt

A Unity rendszerén belül a unit tesztelés célja, hogy külön-külön, izolált módon teszteljük a kód egyes részeit, és biztosítsuk, hogy azok a várt módon működjenek. A unit tesztek lehetővé teszik, hogy a fejlesztők gyorsan és hatékonyan azonosítsák a hibákat a kód egyes funkcióiban anélkül, hogy az egész rendszert végig kellene futtatniuk. A Unity saját unit tesztelési rendszert biztosít, amely a Test Runner eszközként ismert, és amely lehetővé teszi a fejlesztők számára, hogy különálló tesztkészleteket készítsenek, valamint automatikusan lefuttassák azokat a kód változásainak ellenőrzésére.

A unit tesztelés Unity-ben a TestManager eszközzel történik, amely lehetővé teszi egy külön assembly (kódcsomag) létrehozását kifejezetten a tesztek számára. Ez a külön assembly biztosítja, hogy a tesztek izolált környezetben fussanak, és nem befolyásolják a játék egyéb működő elemeit. A tesztek lehetnek például a logikai egységek, algoritmusok vagy adatkezelési funkciók ellenőrzésére, mindezt anélkül, hogy az éles játékfunkciók működését zavarnánk [38].

4.3. Teszt jegyző könyv

A unit tesztelési folyamatok során alaposan ellenőriztem a játékos és az ellenségek mechanikáit, hogy megbizonyosodjak azok helyes működéséről. Az alábbiakban bemutatom a különböző teszteket, amelyek a játékos mozgására, támadásaira, és az ellenségek viselkedésére vonatkoznak.

- **Játékos mozgás tesztek:** A tesztek biztosítják, hogy a játékos megfelelően mozogjon, az irányváltás helyesen történjen, és az ugrás megfelelően növelje a függőleges sebességet. Az IsGrounded és Jump tesztek az alapvető mozgást, míg a Flip teszt az irányváltást ellenőrzi
- **Játékos támadás tesztek:** A tesztek a támadásokat és a kombinációkat ellenőrzik, például a ComboStep biztosítja a támadások sorrendiségét, míg a BlockingBullet teszt a blokkolás működését figyeli.
- **Skorpió mozgás tesztek:** A skorpió mozgását és viselkedését vizsgálják, beleértve a járőrözést, a játékos célzást és a fordulást, hogy megfelelően reagáljon a játékos közeledésére.

A tesztelési folyamatok átláthatóságának növelése érdekében jegyzőkönyvet készítettem a végrehajtott unit tesztekéről. Ez a dokumentáció segíti a tesztek könnyebb nyomon követhetőségét és értelmezését, valamint biztosítja, hogy a tesztelési eredmények megfelelően rögzítésre kerüljenek.

Teszt neve	Eredmény	Megjegyzés
BlockingBullet_SetsBlockingState	Sikeres	Megállítja a lövedéket
ComboStep_IncrementsCorrectly	Sikeres	Helyesen működik
SimulateMoveRight	Sikeres	Helyesen vált irányt
IsGrounded_ReturnsFalse_WhenNoGround	Sikeres	Igaz-at ad ha a talajon áll
Jump_WhenGrounded_ShouldIncreaseYVelocity	Sikeres	Ugrani csak talajról tud
FlipTowardsPlayer_FlipsCorrectly	Sikeres	Helyesen fordul
Patrolling_MovesTowardsNextPoint	Sikeres	Helyesen járőrözik
TargetingPlayer_StopsWhenCloseEnough	Sikeres	Mozgása megáll

4.1. táblázat. Teszt jegyzőkönyvből kiragadott pár teszt leírása

4.4. Összefoglaló

A playtestelés rendkívül fontos szerepet játszott a fejlesztési folyamatban, mivel nemcsak a játékmenet logikai aspektusait, hanem a felhasználói élményt is képes voltam tesztelni és finomhangolni. A playtestelés során számos olyan problémára derült fény, amelyet a szigorúan logikai tesztelés nem feltétlenül tár fel. Például, bár a játék mechanikai elemei jól működtek az elméleti tesztek során, a valós játékelmény során más szempontok is fontosak: a sebesség, az irányítás érzékenysége, az animációk folyamatosága, vagy az, hogy a játékosok hogyan érzékelik a környezetet és az ellenségeket. A playtestek lehetőséget adtak arra is, hogy az esetleges unalmas vagy túl nehéz szakaszokat észrevegyem, és olyan finomhangolásokat végezzek, amelyek segítik a játék gördülékenyebbé tételét.

A Unit tesztek megírása Unityben sokkal nagyobb kihívásnak bizonyult, mint azt előzetesen gondoltam. Bár a tesztelés elmélete egyszerűnek tűnt, a gyakorlatban számos bonyodalom merült fel, amelyek a fejlesztés során felmerülő komplexitásokkal és a Unity specifikus működésével kapcsolatosak.

Összegzés

A szakdolgozatom fejlesztése során rengeteget tanultam a Unity használatáról, különösen a játékmechanikák és interakciók kialakítása terén. Majdnem minden mérföldkövet sikerült teljesítenem, beleértve a játéklogika fejlesztését, az ellenség viselkedésének programozását és az irányítás kialakítását. Jelenleg már csak a grafika integrálása van folyamatban, amely a projekt vizuális aspektusainak finomhangolását és az esztétikai élmény megteremtését célozza.

4.5. Bővítési terv

A projekt következő bővítési fázisa a játék vizuális és mechanikai rendszerének továbbfejlesztésére irányul. A vizuális elemek integrálása jelenleg folyamatban van, miközben a játékmenet dinamikáját is szeretném gazdagítani. A következő lépésben egy olyan boss implementálása szerepel, amely egyszerű AI-t alkalmaz, képes lenne tanulni a játékos viselkedéséből, és ennek megfelelően alkalmazkodni a harc során. Ezáltal a játék kihívása folyamatosan változna, és még izgalmasabb élményt nyújtana a játékosok számára.

Fejlesztetni szeretném a játékban lévő ellenségeket is, mivel úgy érzem, hogy jóval több potenciál rejlik bennük. Az ellenségek viselkedését, támadási mintáit és AI-ját is tovább lehetne finomítani, hogy változatosabbá és kihívásokkal telibbé váljanak. Ezzel nemcsak a játékmenet élvezhetőségét növelném, hanem a játékosok számára új taktikai lehetőségeket és izgalmasabb összecsapásokat biztosítanék. A fejlesztés során a különböző ellenségek egyedi jellemzőit is figyelembe venném, hogy minden típus egyedi kihívást jelenthessen.

Leadott forráskód elérhetősége: github.com/bartusjani/W5OLP9_szakdolgozat

Jelenlegi forráskód elérhetősége: [/github.com/bartusjani/Szakdogajelenlegihelyzet](https://github.com/bartusjani/Szakdogajelenlegihelyzet)

Játék elérhetősége: jankopepe.itch.io/diploma-game

Bemutató videó elérhetősége: youtu.be/SSNWiU138hQ

Ábrák jegyzéke

1.1. ábra –saját ábra.

1.2. ábra–saját ábra.

Irodalomjegyzék

- [1] GODOT ENGINE: *GScript*, elérhető: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html [Letöltve: 2025-04-10]
- [2] GODOT ENGINE: *Version 1.0*, elérhető: <https://godotengine.org/article/godot-engine-reaches-1-0/> [Letöltve: 2025-04-10]
- [3] GODOT ENGINE: *Version 2.0*, elérhető: <https://godotengine.org/article/godot-engine-reaches-2-0-stable/> [Letöltve: 2025-04-10]
- [4] GODOT ENGINE: *Version 3.0*, elérhető: <https://godotengine.org/article/godot-3-0-released> [Letöltve: 2025-04-10]
- [5] GODOT ENGINE: *Version 4.0*, elérhető: <https://godotengine.org/article/godot-4-0-sets-sail> [Letöltve: 2025-04-10]
- [6] GODOT ENGINE: *Licences*, elérhető: https://docs.godotengine.org/en/stable/about/complying_with_licenses.html [Letöltve: 2025-04-10]
- [7] UNREAL ENGINE: *Blueprint*, elérhető: <https://dev.epicgames.com/documentation/en-us/unreal-engine/coding-in-unreal-engine-blueprint-vs.-cplusplus> [Letöltve: 2025-04-10]
- [8] UNREAL ENGINE: *Blueprint classes*, elérhető: <https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-blueprints-visual-scripting-in-unreal-engine> [Letöltve: 2025-04-10]

- [9] UNREAL ENGINE: *Version 4.27*, elérhető: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-4.27-release-notes?application_version=4.27 [Letöltve: 2025-04-10]

- [10] UNREAL ENGINE: *Version 5.0*, elérhető: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5.0-release-notes?application_version=5.0 [Letöltve: 2025-04-10]

- [11] UNREAL ENGINE: *Version 5.3*, elérhető: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5.3-release-notes?application_version=5.3 [Letöltve: 2025-04-10]

- [12] UNREAL ENGINE: *Version 5.5*, elérhető: <https://www.unrealengine.com/en-US/blog/unreal-engine-5-5-is-now-available> [Letöltve: 2025-04-10]

- [13] UNREAL ENGINE: *Features*, elérhető: <https://www.unrealengine.com/en-US/features> [Letöltve: 2025-04-10]

- [14] UNREAL ENGINE: *Licences*, elérhető <https://www.unrealengine.com/en-US/license> [Letöltve: 2025-04-10]

- [15] MICROSOFT: *C# language documentation*, elérhető: <https://learn.microsoft.com/en-us/dotnet/csharp/> [Letöltve: 2025-04-10]

- [16] UNITY: *Version 4.0*, elérhető: <https://unity.com/news/unity-4-0-launches> [Letöltve: 2025-04-10]

- [17] UNITY: *Version 5.0*, elérhető: <https://unity.com/news/unity-5-here> [Letöltve: 2025-04-10]

- [18] UNITY: *Version 6*, elérhető: <https://unity.com/releases/unity-6-releases> [Letöltve: 2025-04-10]

- [19] UNITY: *Licences*, elérhető: <https://unity.com/products/compare-plans> [Letöltve: 2025-04-10]

- [20] UNITYSCENE: *Creating scenes*, elérhető: <https://docs.unity3d.com/Manual/CreatingScenes.html> [Letöltve: 2025-04-10]
- [21] UNITYGAMEOBJECT: *GameObject*, elérhető: <https://docs.unity3d.com/6000.0/Documentation/Manual/class-GameObject.html> [Letöltve: 2025-04-10]
<https://docs.unity3d.com/Manual/Components.html>
- [22] UNITY: *Introduction to components*, elérhető: <https://docs.unity3d.com/Manual/Components.html> [Letöltve: 2025-04-10]
- [23] UNITY: *Transforms*, elérhető: <https://docs.unity3d.com/Manual/class-Transform.html> [Letöltve: 2025-04-10]
- [24] UNITY: *Collider2D*, elérhető: <https://docs.unity3d.com/Manual/2d-physics/collider/collider-2d-landing.html> [Letöltve: 2025-04-10]
- [25] UNITY: *Introduction to RigidBody 2D*, elérhető: <https://docs.unity3d.com/Manual/2d-physics/rigidbody/introduction-to-rigidbody-2d.html> [Letöltve: 2025-04-10]
- [26] UNITY: *Sprite renderer*, elérhető: <https://docs.unity3d.com/Manual/sprite/renderer/renderer-landing.html> [Letöltve: 2025-04-10]
- [27] UNITY: *Introduction to scripting*, elérhető: <https://docs.unity3d.com/Manual/intro-to-scripting.html> [Letöltve: 2025-04-10]
- [28] UNITY: *Lighting*, elérhető: <https://docs.unity3d.com/Manual/Lighting.html> [Letöltve: 2025-04-10]
- [29] UNITY: *Prefabs*, elérhető: <https://docs.unity3d.com/Manual/Prefabs.html> [Letöltve: 2025-04-10]
- [30] UNITY: *Using the scene view*, elérhető: <https://docs.unity3d.com/Manual/UsingTheSceneView.html> [Letöltve: 2025-04-10]
- [31] UNITY: *Game view*, elérhető: <https://docs.unity3d.com/Manual/GameView.html> [Letöltve: 2025-04-10]
- [32] UNITY: *Animator Controller window*, elérhető: <https://docs.unity3d.com/Manual/class-AnimatorController.html> [Letöltve: 2025-04-10]


- [33] UNITY: *Inspector window*, elérhető: <https://docs.unity3d.com/Manual/UsingTheInspector.html> [Letöltve: 2025-04-10]
- [34] UNITY: *Hierarchy window*, elérhető: <https://docs.unity3d.com/Manual/Hierarchy.html> [Letöltve: 2025-04-10]
- [35] UNITY: *Project view*, elérhető: <https://docs.unity3d.com/Manual/ProjectView.html> [Letöltve: 2025-04-10]
- [36] UNITY: *Animation window*, elérhető: <https://docs.unity3d.com/Manual/AnimationEditorGuide.html> [Letöltve: 2025-04-10]
- [37] UNITY: *Animator component*, elérhető: <https://docs.unity3d.com/Manual/class-Animator.html> [Letöltve: 2025-04-10]
- [38] UNTIY: *Test framework introduction*, elérhető: <https://docs.unity3d.com/6000.2/Documentation/Manual/test-framework/test-framework-introduction.html> [Letöltve: 2025-04-10]
- [39] UNTIY: *Event functions*, elérhető: <https://docs.unity3d.com/Manual/event-functions.html> [Letöltve: 2025-04-14]
- [40] UNTIY: *Late Update*, elérhető: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html> [Letöltve: 2025-04-14]
- [41] GITHUB: *About GitHub and Git*, elérhető: <https://docs.github.com/en/get-started/start-your-journey/about-github-and-git> [Letöltve: 2025-04-10]
- [42] ATLISSIAN: *Trello*, elérhető: <https://trello.com/guide> [Letöltve: 2025-04-10]

Nyilatkozat

Alulírott *Bartus János*, büntetőjogi felelősségem tudatában kijelentem, hogy az általam benyújtott, *2D platformer játék fejlesztése Unity keretrendszerben* című szakdolgozat önálló szellemi termékem. Amennyiben mások munkáját felhasználtam, azokra megfelelően hivatkozom, beleértve a nyomtatott és az internetes forrásokat is.

Aláírással igazolom, hogy az elektronikusan feltöltött és a papíralapú szakdolgozatom formai és tartalmi szempontból mindenben megegyezik.

Eger, 2025. április 14.


aláírás